Universitatea POLITEHNICA din Bucure[U+0219]ti
Facultatea de Automatic[U+0103] [U+0219]i Calculatoare
Departamentul Calculatoare

Lucrare de Diplom[U+0103]

Algoritm de rutare pentru conectori ortogonali

Autor

Mu[U+0219]at Andrei-Alexandru

Coordonatori:
As. Drd. Ing. Andrei Voinescu
As. Drd. Ing. Dan Dragomir

Bucure[U+0219]ti, Septembrie 2014

University POLITEHNICA of Bucharest
Faculty of Automatic Control and Computers
Computer Science and Engineering Department

Diploma Thesis

Routing algorithm for orthogonal connectors

Author

Mu[U+0219]at Andrei-Alexandru

Supervisors:
As. Drd. Ing. Andrei Voinescu
As. Drd. Ing. Dan Dragomir

Bucharest, September 2014

Contents

Abstract

Diagrams are used in a variety of fields such as hardware design and verification as a means to represent workflows and concepts. They require dedicated algorithms in order to produce easy to understand results. Currently available software is clunky and, at times, it can be really slow and will often result in layouts which are hard to read and understand. The main issue of layout algorithms is the sheer amount of data that a graph can contain, which makes placing and routing diagrams in a limited space a very difficult problem. We solve this problem by using a two phase algorithm which tries to generate layouts which feel natural for the user. The first phase will layout the diagrams using a genetic algorithm and a set of pre-determined constraints, while the second phase will route the paths connecting the diagrams elements using orthogonal routing. With this approach, we can provide layouts which suit the needs and preferences of any user, instead of providing generic representations.

Keywords Diagrams, Layout Processing, Orthogonal Routing, Genetic Algorithms, Hardware Design

Chapter 1

Introduction

Diagrams are tools which are widely used in various domains in order to represent concepts, hierarchies, architectures etc. In software engineering, they can be found as an UML(Unified Modeling Language) visualisation of a system or program or as an inheritance tree in the case of OOP(Object Oriented Programming) languages. Diagrams are also present in the hardware world where their usage includes: logical representation of a FSM (Finite State Machine), data flow between components or on-chip dependencies between components. However, there is a field in which diagrams are a key part of development: hardware design and verification. In the case of hardware design, be it ASIC (Application Specific Integrated Circuit), FPGA (Field Programmable Gate Array), processors, memories or microcontrollers, diagrams are used to describe the individual blocks which are put together to form the respective chip, to represent the way modules are connected at a logical level or to show how data is passed and manipulated by each component. In hardware verification, they are used to represent hierarchies and inheritance amongst specific units such as agents, sequencers, test scoreboards and general test flows, often corresponding with the guidelines of accepted methodologies such as OVM (Open Verification Methodology) and UVM (Universal Verification Methodology). Therefore, one can assert that diagrams are instrumental in all stages of hardware design and verification.

Despite being an important part of the hardware design and verification world, diagrams do not receive the required attention from the developer community. While HDLs (Hardware Description Languages) have been gaining popularity over the years and IDEs have been developing to support them, diagrams have not received adequate attention from the community. It is often overlooked that their main utility lies in understanding and maintaining the coherence and logical flow of a design or a verification testbench.

Usually, a hardware design consists of a number of design elements which can range from hundreds to thousands of components; the equivalent of a graph with

the same number of nodes. Representing such a design as a diagram would be, in turn, equivalent to drawing said graph in a plane. A given graph is transformed into a diagram representation through a three step process: planarity testing, node placement and connection routing. This is a well known problem which is difficult to solve with current algorithms and software in a reasonable amount of time and also yield a readable and easy to understand result. However, there are methods and algorithms which can be combined to obtain a feasible diagram representation.

The layout and routing solution which is presented in this study combines techniques used in current solutions with newer algorithms in an attempt to obtain representations which respect the following criteria:

- Coherency: The diagram should be easy to understand and follow. Each connection in the diagram represents either data paths (BUS) or logical connections between components. For this reason the user must be able to understand where these paths start, where they end and what components they connect.
- Correctness: As any algorithm, the solution must maintain the correctness of the model when representing it. All nodes have to be included and all edges represented according to the corresponding adjacency matrix of the given graph.
- Planarity: Refers to the property of the given graph to be represented in the two dimensional plane. A planar graph does not have intersecting edges. It is expected that the representation preserves the planarity property of the graph.
- Orthogonality: Edges between nodes are represented as orthogonal connectors. Orthogonality implies that a path is composed only of perpendicular segments and contains a minimum number of bends. Paths shall not overlap any node.
- Performance: The proposed implementation has to produce the expected results in an acceptable amount of time. One of the biggest concerns regarding the types of algorithms which operate on graphs is the amount of time they take to finish analysis and output their results.

Chapter 2

Related Work

Even though graph embedding and diagram drawing is a popular subject which has been the topic of articles and research, actual open source software which can do such operations is scarce. Algorithms which solve embedding problems have been described and implemented since the year 1970. The major impediment for any application which would like to perform diagram drawing is actually the diversity and ambiguity of the problem. Apart from performance analysis (time and memory consumption), and corectness of the drawing, any other aesthetic appreciation of the result is subjective. Depending on who the diagram is addressing and what is its final scope, certain styles, connections and layouts are appropriate, while others render the final result unusable.

This does not mean that a middle-ground approach does not exist and that there are no techniques or conventions which are generally accepted as good practice. There are two commonly used open source libraries which provide support for drawing diagrams: Graphviz dot[?] and Eclipse GEF[?].

2.1 Graphviz dot

Graphviz dot is, as described in its documentation, a command line tool which draws directed graphs as hierarchies. It reads its information from text files written in the DOT language and outputs the drawing in graphic formats such as GIF (Graphics Interchange Format), PNG (Portable Network Graphics), SVG (Scalable Vector Graphics) or PDF (Portable Document Format).

The DOT language input file specifies general information describing the graph. It accepts the definition of three main objects: graphs, nodes and edges. A simple graph (listing **??**) is described only by its name and a list of nodes and edges. If the user desires, subgraphs can be defined inside the graph with distinctive characteristics and properties. The same can be done for nodes and edges (listing **??**).

Listing 2.1: Simple dot file format

```
digraph G {
    A -> B -> C;
    A -> D;
    A -> E;
    C -> F;
    C -> G;
    D -> F;
    A -> G;
    C -> H;
}
```

In terms of implementation, *dot* uses methods such as force directed placing to layout its graphs. Also, its layout algorithm presumes graphs are acyclical. This is why *dot* ensures that all cycles are removed from a graph before sending it to the layout routine.

Listing 2.2: Customized graph dot file representation, as can be seen in the official documentation

```
digraph G {
    size ="4,4";
    main [shape=box]; /*this is a comment*/
    main -> parse [weight=8];
    parse -> execute;
    main -> init [style=dotted];
    main -> cleanup;
    execute -> { make_string; printf}
    init -> make_string;
    edge [color=red]; // so is this
    main -> printf [style=bold,label="100 times"];
    make_string [label="make a\nstring"];
    node [shape=box,style=filled,color=".7 .3 1.0"];
    execute -> compare;
}
```

As user-oriented features, the drawings can be manipulated and customized by the user either from a graphical interface or by directives specified in the DOT input file. A few examples of such features are: setting dimensions and clearance values for node positioning, specifying the direction of edges, label nodes, creating hyperlinks.

Since it is a command line tool, *dot* can be integrated with many applications and development environments. From custom applications to IDEs, various implementations can use this tool as a means to support drawing diagrams (figure **??**).

The only requirement is that such an application posseses a means to generate the dot configuration file.

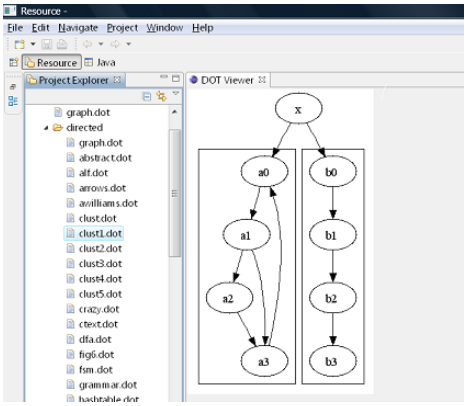

Figure 2.1: An example of graphviz integrated with the Eclipse IDE.*

2.2 Eclipse GEF

Eclipse GEF (Graphical Editing Framework) provides support for creating graphical editors and views in the Eclipse IDE environment[?]. The graphical editors created using GEF allow the user to mainly hand draw any desired graph (figure ??). All the basic elements required for the task are predefined by the underlying library and routing can be performed manually.

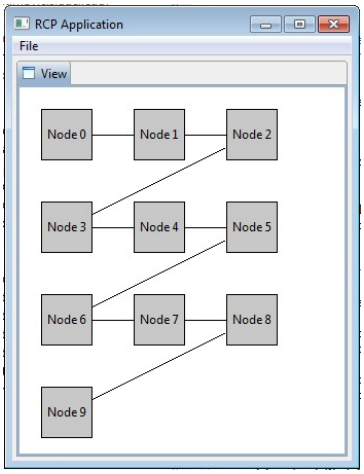

Figure 2.2: Example of a graph viewer implemented using the GEF library.†

*Image taken from http://blog.abstratt.com/
†Image taken from http://www.programcreek.com/

Above this basic level, GEF contains classes specialised in layout and connection routing. By default, the library has implementations for layout algorithms such as grid layout and tree layout. However, these classes expose APIs which can be used to create new and more complex layout implementations.

A notable drawback of the GEF library is the lack of documentation for its API. This may be the reason why it is not widely adopted, even amongst the Eclipse community. The main sources of information is from articles or community made tutorials, such as those of Lars Vogel[?]. Another way of familiarizing oneself with the API and experiencing its capabilities is by running and experimenting with the examples suite which comes with the library. These examples showcase typical situations and functionalities through simple widgets. While they are educational and helpful for developers, they require an investment of time in order to understand the showcased code and functionality.

An extension of Eclipse GEF is Eclipse Zest[?] (figure ??), which is a visualization toolkit with implementations for different types of graph viewers. In essence, a graph viewer is a layout algorithm which specializes in a certain representation of the graph. As specified on their homepage, Zest includes a predefined layout package containing algorithms for the following types of layouts: Spring, Tree, Radial and Grid. Zest is open to contributions from the community regarding new types of layouts. Unlike GEF, Zest is does not give the same amount of liberty regarding the representation of nodes. They are static entities, with fewer available customizations. Nodes can contain only minimal decorations such as text and an image. This shows that the emphasis of this library is on the layout itself.

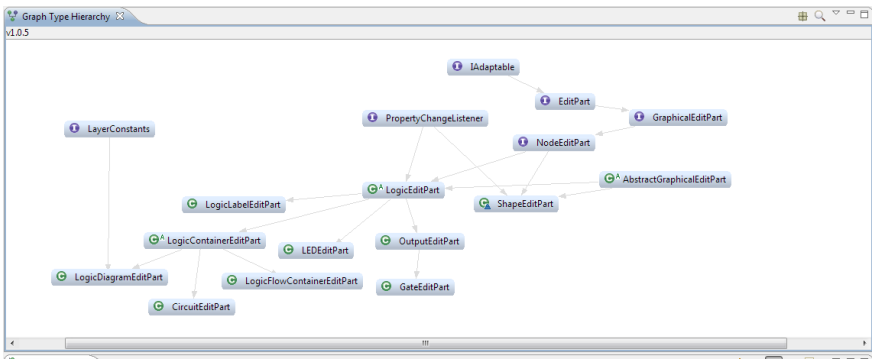

Figure 2.3: A graph viewer using Zest radial layout.[‡]

[‡]Image taken from http://pbwhiteboard.blogspot.ro/

Chapter 3

Algorithm Design

This chapter presents how the application is designed at a logical level and describes the main algorithms and techniques which are combined and used by the implementation.

3.1 Planarity testing

The first algorithm used by the application is the planarity testing method known as "vertex addition". In order to properly present this algorithm, we must first introduce the necessary notions regarding graph planarity and methods of testing this property.

3.1.1 Planarity property and criteria

A given graph *G=(V,E)* is planar if it can be drawn on a plane and its edges never cross each other, i.e. they intersect only at their endpoints. This type of drawing is also known as a planar embedding of the graph.

In order to determine if a graph possesses the planarity property, a series of theorems and criteria have been stated over the years. Amongst the first of these criteria is a theorem published by the Polish mathematician Kazimierz Kuratowski [?] in 1930. The theorem deals with subdivisions, i.e. graphs which result from inserting vertices into edges. It states that a planar graph shall not contain a subdivision of the forbidden graphs K5(the complete graph on five vertices) or K3,3(complete bipartite graph on six vertices, three of which connect to each of the other three, also known as the utility graph). It is formulated as follows:

A finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K5 or K3,3 .

Another important theorem which deals with planarity was formulated by the German mathematician Klaus Wagner. This theorem takes into consideration graph minors instead of subdivisions. A graph H is called a minor of a given graph G if H is obtained by deleting vertices and edges or by contracting edges. The Wagner theorem states that a finite graph is planar if and only if it does not have K5 or K3,3 as a minor.

While these theorems manage to correctly define the planarity problem in a mathematical way, they are not optimal criteria to use in practice. The main reason is efficiency: we would like the complexity of such an algorithm to be linear - O(n). In practice, there are other theorems and criteria which fit in the linear complexity. For example, given a finite, connected planar graph with v the number of vertices, e the number of edges and f the number of faces (regions bounded by edges, including the outer, infinitely large region), the following hold true:

Theorem 1: If $v \geq 3$ *then* $e \leq 3v - 6$*;*
Theorem 2: If $v \geq 3$ *and there are no cycles of length 3, then* $e \leq 2v - 4$*;*
Theorem 3: $v - e + f = 2$ (Euler's formula).

Unfortunately, these theorems are only necessary conditions, not sufficient conditions. They can only be used to prove that a graph is not planar; they cannot prove that a graph is planar.

3.1.2 Vertex addition method

The edge addition algorithm is the result an intensive research started in the 1960s by Abraham Lempel, Shimon Even and Cederbaum. They created an algorithm to determine whether a graph is planar or not and embed it in the plane in O(n^2) time. Later on, Even and Tarjan developed a method to generate the st-numbering of a graph in linear O(n) time.

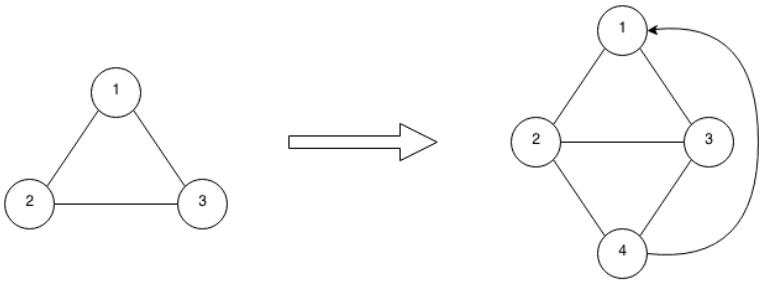

Figure 3.1: St-numbering representation of the G(3,3) graph

An st-numbering[?] of a graph G is a subgraph H which has a set of properties, as follows. First, nodes in G are numbered. Then the vertex with the lowest number is designated as source vertex, *s*, and the one with the highest number is designated

target, *t*. All the remaining vertexes which are not the source or target must be connected to a vertex numbered lower than them, and a vertex numbered higher than them (figure **??**). In order to achieve this configuration, the subgraph H may add additional nodes to the graph G.

Finally, Kellogg Booth and George Lueker created a data structure which represents the possible embeddings of a graph (or, in practice, induced subgraphs of a graph) called a *PQ-tree*. Using this data structure and the previous methods, they managed to created a linear time algorithm to determine planarity and possible embeddings. We will next present the general steps of an implementation of the Vertex addition method as proposed by Norishige Chiba and Takao Nishizeki[?].

First, it computes the st-numbering of the given graph. It then creates a PQ-tree which contains only one P node, the source, and all the other nodes are leaves. Next, a loop is entered which, for every leaf node, shall perform two steps:

- A reduction step which attempts to gather all matching leaves into a P node which respects the st-numbering. If this step fails, then the graph is not planar.
- A vertex addition step, in which full nodes are replaced by a single P node and all the neighbours of that node numbered higher than itself are added as leaves.

Listing 3.1: Vertex addition algorithm as proposed by Chiba and Nishizieki

```
1  PROCEDURE PLANAR(G);
2  BEGIN
3  st_numbering(G);
4  make_PQ_tree(s); //construct PQ-tree with root in s
5  FOR v := 2 to n:
6      BEGIN
7          // start of reduction step
8          reduction = gather_leaves(template, subtree_root);
9          IF reduction == FAILED:
10         THEN BEGIN
11             set_planar(G) = FALSE;
12             RETURN
13         END
14         // end of reduction step
15
16         // start of vertex addition step
17         replace_full_node(PQ-tree, P-node)
18         FOR t in neighbours(v):
19             BEGIN
20                 IF st_number(t) > st_number(v):
```

```
21                  THEN BEGIN
22                      add_to_PQ_tree(t);
23                  END
24              END
25          // end vertex addition step
26      END
27  set_planar(G) = TRUE;
28  END
```

Figures 3.2 to 3.4 below present the steps of this algorithm applied on a graph G(3,3).

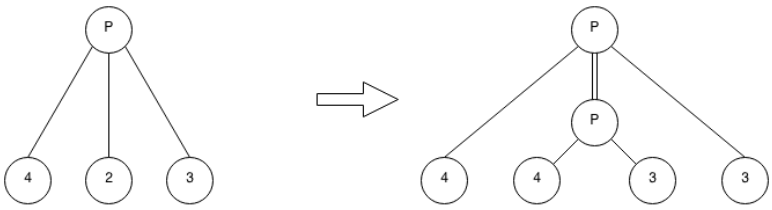

Figure 3.2: The initial representation of the graph and the configuration after one merge step

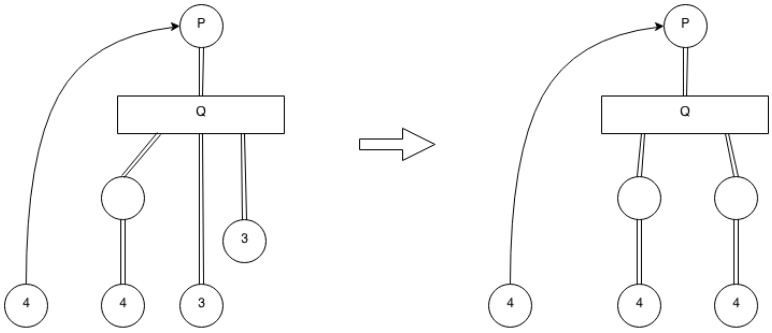

Figure 3.3: Graph representation after two more reduction steps.

3.2 Graph placement

Graph placement refers to assigning a position (set of coordinates) to each node of the graph in the space where the graph has to be represented.

We saw in the previous chapter that algorithms which determine if a graph is planar can also generate its embeddings. However, this is not the same as actually drawing the graph. In reality, we are constrained by the limitations of the space in which we want to draw the diagram. Therefore, even though we know that a possible arrangement of the nodes will allow us to represent the graph in a plane, we cannot

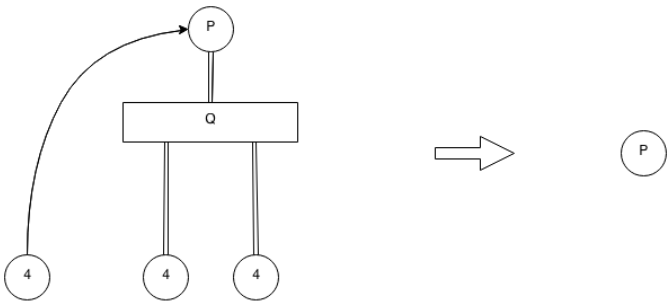

Figure 3.4: Complete PQ-tree representation and the final merge into a single P node

be certain that the representation is feasible or understandable. Furthermore, one may want to draw even graphs which are not planar, knowing that two or more of its edges will cross one another.

For this reason, a logical positioning of each node in the given plane is required. This step greatly facilitates the routing process and also helps reduce the time and complexity of said operation.

Commonly used methods which are also incorporated in the solution proposed by this thesis include grid placement and force directed graph-drawing. We shall continue by presenting these methods in the following sections.

3.2.1 Grid placement

Grid placement[?] is one of the most popular and most used techniques by diagram-drawing algorithms. It splits the space in which the representation is done into a grid and then starts placing the nodes in a certain order. Different approaches can be found here, depending on the type of the graph.

Trees are often placed in layers. The root of the tree is first fixed on an edge of the grid, and then each level of the tree visited in breadth-first order forms a layer of nodes. Layers can be arranged vertically or horizontally, depending on the implementation.

More general graphs may also follow this pattern, by computing the spanning tree[?] of the graph and placing nodes as stated above, starting with the spanning tree's root. Other approaches for general graphs may place and pin the node with the highest number of connections in the middle of the grid.

The advantages of this method are represented by its ease of implementation, small consumption of resources and low complexity. A main drawback is that it generally interferes with the routing process and does not facilitate it. It also poses the risk of drawing planar graphs as though they were non-planar, i.e. intersecting various edges.

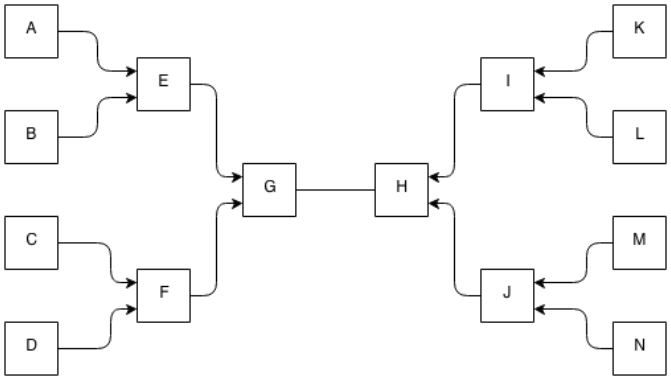

Figure 3.5: Example of a tournament structure organized as a grid.

3.2.2 Force directed graph-drawing

Force directed drawing[?] is a technique which borrows concepts from physiques. In order to apply this method, one must model the graph as a physical system in which elements (nodes) interract with each other through forces. Connected nodes will be pulled towards each other by attraction forces (such as spring forces base on Hooke's Law), while unrelated nodes reject each other (similar to electrically charged particles according to Coulomb's Law). These forces are applied continuously until the system reaches stability.

Using such a technique will yield a graph placement which has the strongly connected nodes pulled in the middle of the assigned drawing spaced, while the individual or loosely connected nodes are pushed towards the edges. The same holds if the graph is split into multiple independet connected components: the nodes of each component shall be pulled towards each other, while the individual components will reject one another.

This approach is better suited to ease the routing process and reduce its time and complexity. It is less likely that planar graphs shall be improperly drawn and routes will be shorter and clearer. However, unlike Grid placement, it is an iterative process which may take considerable amounts of time to reach stability, while also consuming more resources.

3.3 Edge routing

The final step in drawing a diagram or a graph is represented by edge routing. Its purpose is to highlight the connections and dependencies between nodes via a set of lines which add up to make paths between nodes. An important aspect is that paths must not cross over other nodes, even though this may increase the length of a path

*Image taken from http://wonderfl.net/

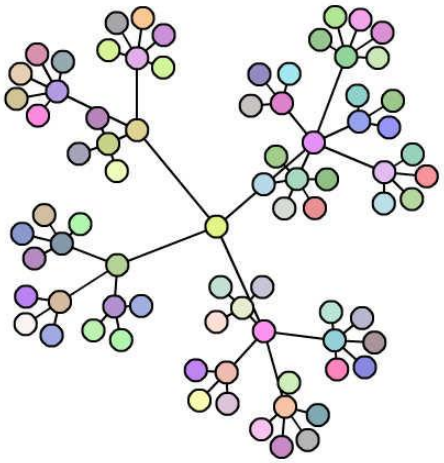

Figure 3.6: A force-directed graph layout with a central node and multiple connected components.*

by adding extra segments to it. Depending on how the paths are represented, edge routing may be:

- Shortest path routing: Nodes are connected only by straight which go directly from the source to the destination. In some implementations, it is acceptable for these lines to cross other nodes, while in others extra segments are added in order to go around obstacles.
- Arc routing: This approach represents paths as arcs. Nodes are usually placed in a linear layout (all nodes are placed on the same axis) and each edge is considered a diameter line for a circle. Then an arc of corresponding length is drawn to join the pair of nodes.
- Orthogonal routing: In orthogonal routing, paths are composed only of vertical and horizontal lines, joined by 90 degree bends. While it is not the most efficient method with regards to the space occupied by the drawing, it does provide clear and direct paths which do not cross over other elements.

3.3.1 Rule-based genetic routing algorithm

Having presented the range of algorithms which help achieve the goal of representing and drawing diagrams, we can now explain and exemplify how the implementation of this thesis works. We shall go through each of the three steps: planarity testing, node placement and edge routing, explaining in detail the design of each level.

The proposed implementation starts by testing the planarity of the input graph using the criteria derived from Euler's theorem. These theorems are applied first be-

cause of their reduced complexity and because all the needed information is computed when the graph is first constructed. Should the result be negative, there is sufficient information to decide that the graph is not planar and that the drawing will have crossing edges. If, otherwise, the result is positive, we know that the theorems are only a necessary condition, not a sufficient one. At this point, the PQ-tree algorithm is applied to thoroughly decide if the graph is planar.

With the planarity property of the graph decided, the algorithm proceeds to the placing step. Here, the nodes will be assigned positions in the drawing space (the canvas) based on certain criteria. This can be achieved in two ways: if the graph is a planar graph with the number of nodes below a specified threshold, the layout shall be a grid. Otherwise, the main routing algorithm is applied. This selection of layout models ensures that the application does not complicate a problem which presents an easy solution.

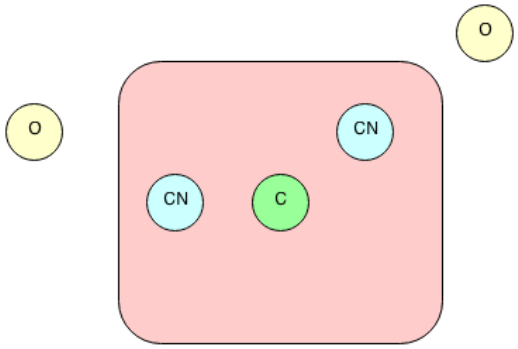

Figure 3.7: The reserved area around the central (C) node. Its neighbours (NC) are permitted inside, but other nodes (O) are not.

In the case of the rule-based algorithm, during the first step, as seen in grid placing, the node with the highest number of connections is placed in the center of the canvas. Then, a certain amount of canvas space is reserved around this node. The central node's neighbours shall be placed in this area in a random order, and no other node may be placed in this area. This step ensures that the random factor is mostly eliminated for this group of nodes and a certain consistency is kept amongst consecutive drawings of the same graph. All the other nodes are placed randomly in the remaining area.

At this point, notions from genetic algorithms are introduced. The graph is considered a population and each node represents an individual. For the remaining steps, each individual shall be evaluated by a fitness function and suffer modifications depending of the result. In order to change and evolve the population, the position of a node shall be modified in the recombination step. Each node outside the reserved area shall generate, along with its neighbour closest to the center, a new pair of nodes which are either closer to each other (attracted) or further apart from each

other (rejected), as described by the force-directed method. This step shall continue until a certain number of steps has been exhausted (we have explored the maximum allowed samples of populations) or the system has reached stability.

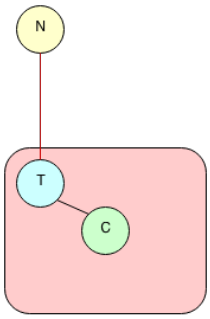

Figure 3.8: Node N does not satisfy the distance threshold.

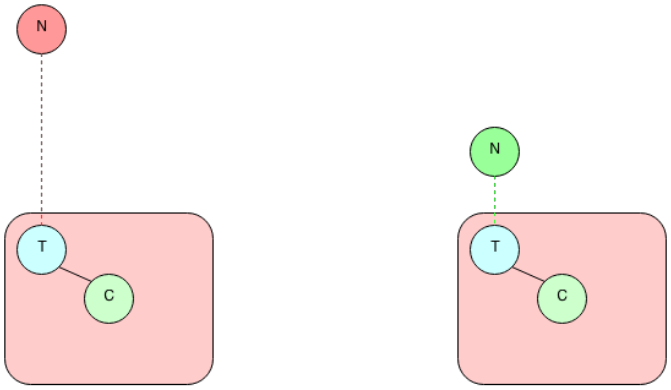

Figure 3.9: The possibilities for N to be moved. The right image is the better solution.

To easily keep track of which rules a node is currently violating as well as facilitate the computation of fitness, each node keeps track of a byte field. The field acts as a register, each bit representing a certain rule. When the node violates a rule, the corresping is set to one. The usage of this register structure relies on the fact that the number of rules is limited.

The last step is the edge routing. The method used is orthogonal edge routing. However, the process starts as a shortest path routing. The initial paths are direct lines between two nodes. Then, the paths are checked to see if they do not cross any obstacles. Should it cross another node, the path is split into orthogonal segments which go around the obstacle. This step is repeated until the path is clear of all obstacles and is incidental only to the source and destination. Lastly, should the drawn graph be planar, the drawing is checked for edge intersection. Paths which do not respect the planarity property are recomputed.

Chapter 4

System Architecture

In this chapter we shall discuss how data is processed and transformed in order to obtain the information required by the graphical library. We shall highlight the main system components which perform modifications on the graph data, starting with the Tokenizer which reads the input files and ending with the Eclipse Draw-2D API.

4.1 Input file processor

Input graphs are initially described in a source text file in a pseudo programming language (discussed in Chapter ??). These files must be analyzed in order to extract the information regarding edges and vertexes. This is achieved by using a parser.

This parser splits the input file into a list of tokens and then traverses this list, searching for certain keywords and constructs, as specified by the grammar. When a keyword describing a graph element is encountered, subsequent tokens are extracted until the expression for that graph element is completed. Using this expression and the information it contains, a corresponding vertex or edge object is created.

After the parser is finished extracting the data from the input file, it aggregates the vertexes and edges and constructs an object representing the graph. This object shall be the main container of information from now on. It is shared by the other components of the system, but all modifications are perfromed iteratively, never concurrently.

4.2 Planarity tester

After the graph container object is created, the first computational unit notified is the planarity tester. From the data which is provided by the graph container, the tester utilizes at first only the number of edges and vertices to apply the theorems

mentioned in the ?? chapter. Should it fail to prove that the graph is not planar, the complete information regarding vertices and edges is extracted in order to apply the PQ-tree algorithm.

This unit does not modify the graph data in any way. Its result is stored internally and is accessed by the drawing library, which will display it as a relevant message for the user. Once the result has been stored, the layout processor is notified that it can access and perform operations on the container object.

4.3 Layout processor

This is the first unit which performs persistent modifications on the graph data. Its purpose is to determine the final coordinates for each node on the canvas, and mark the nodes as pinned, ensuring that no operation may modify their coordinates as a side-effect. Information about edges is also necessary, but only as an indication for the relationship between nodes.

Initially, the layout processor shall decide, based on the number of nodes, whether a grid layout or the genetic layout is more efficient. Generally, if the number of nodes is below a certain threshold, i.e. the algorithm is presented with a simple graph, it shall be layed out as a grid, otherwise the custom set of rules is applied. Next, the chosen algorithm is applied and the modifications are stored by the graph container object. A notification is then given to the connection router, so that it may begin the operating on the edges.

4.4 Connection router

The last step which must be performed before the graph is ready to be displayed is to compute the sets of points which compose the paths between nodes. The router is responsible with performing this operation.

The connection router analyses the existing edges in the graph container and builds corresponding orthogonal connectors. This is a three-step operation:

- First, all nodes are connected directly using the shortest path method. All connections which are represented by vertical or horizontal lines that do not cross any other element are kept.
- Second, those connections which did not match the criteria from step one are corrected (they are deviated to avoid obstacles). Also, connections which were not orthogonal before are orthogonalized.
- Finally, a performance optimization is performed on connections. Redundant points are removed from paths (for example a segment which contains three points shall have the middle one removed) and paths which can be shortened without breaking the routing rules are modified accordingly.

Once this step is done, the graph container object holds all the data necessary for the drawing. This object shall be passed on to the drawing unit which utilizes the Eclipse draw-2D API.

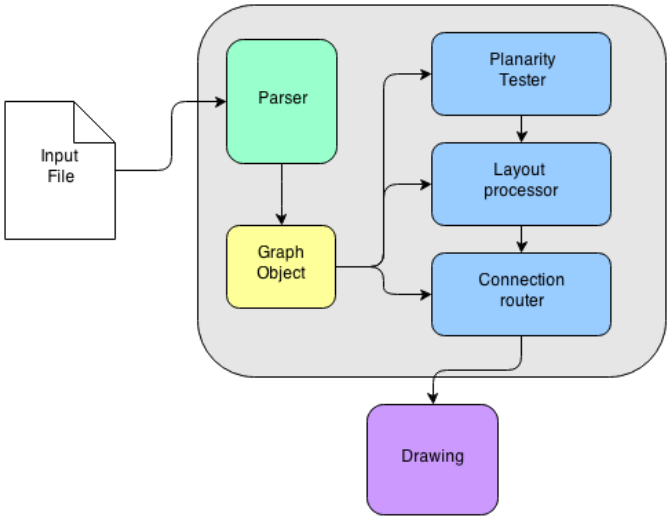

Figure 4.1: Diagram of the system workflow

Chapter 5

Software Implementation

In this chapter we shall discuss the programming techniques, language characteristics and APIs used in implementing each system component and algorithm. We shall highlight the APIs used and the integration with the Eclipse platform, as well as how OOP techniques and patterns are used to implement some of the algorithm presented in chapter ??.

5.1 Eclipse platform and Draw 2D graphics API

The Eclipse platform is an open source, cross-platform IDE(Integrated Development Environment) environment developed and maintained mainly by a consortium which includes companies such as IBM, Red Hat and SuSE. It mainly targets Java developers, but it also supports the plug-in additions to extend its functionalities for various programming languages. It can also work with typesetting languages such as LaTeX and be integrated with revision control systems like CVS(Concurrent Versions System) and GIT.

Eclipse has become popular amongst both software and hardware developers. Its main advantages are: faster code navigation and visualisation, better code understanding through tracing, referencing and hierarchical display functionalities and resource usage management and monitorization. The platform is also able to handle makefile generation, compilation and, most importantly, integration with debugging and profiling tools.

This IDE enviroment has been chosen mainly because of it can operate cross-platform and it is open source. Moreover, the code does not have to be written and organized differently in order to function on different platforms, such as Windows or OS-X. The drawbacks are also relevant, since running any application requires the entire platform to load, which is significantly more taxing on resources than a

command line application would be. Thus, the IDE can be considered suboptimal from this perspective if the user is not fully interested in all of its capabilities.

Since it is an open source project, the Eclipse platform offers developers various APIs to facilitate and encourage the community to extend or improve existing functionality or add new capabilities. One such API is the Draw-2D graphics library, which is an extension of the SWT library. This library is generally used to implement small widgets which handle the drawing of charts and diagrams.

By default, the API can render any object which the SWT library can handle. In addition, it adds a series of classes and methods which allow the user control over the position of each object, its contents and other graphical characteristics such as colour and style. We shall continue by enumerating and discussing some of the main Draw-2D classes used in the implementation.

5.1.1 Draw 2D Point class

Points represent the basic operating unit of the library. They do not refer to a graphical entity, but a spatial one. They are used to define the location (coordinates) of a figure or shape, designate an axis or segment or define the corners of a polygon when placed in an ordered list.

The class exposes API for accessing and manipulating coordinates, as well as basic mathematical computations such as transposing points and calculating the euclidian distance between two points. Still, the API lacked more advanced computations such as: manhattan distance, determining the angle between the X axis and the line designated by two points and moving the point in a general direction with a given speed. These functions have been implemented and are shown below:

Listing 5.1: Functions added for the Point class

```
1  public int manhattanDistance(Point p1, Point p2) {
2      return Math.abs(p1.x - p2.x) + Math.abs(p1.y - p2.y);
3  }
4
5  public double computeAngle(Point p1, Point p2) {
6      int deltaY = p1.y - p2.y;
7      int deltaX = p1.x - p2.x;
8      return (Math.atan2(deltaY, deltaX) * 180) / Math.PI;
9  }
10
11 public void movePointInDirection(Point p, double speed_factor) {
12     p.x += speed_factor * (int) Math.cos(angle);
13     p.y += speed_factor * (int) Math.sin(angle);
14 }
```

5.1.2 Draw 2D Rectangle class

The rectangle in the Draw 2D library does not simply stand for the geometric shape. When drawn, every figure and component placed on the canvas is placed in an enclosing rectangle which delimits its size and relative position by means of five points: the center and four corners.

Whenever testing for collisions, it is not the figures and shapes which are intersected, but rather their enclosing rectangles. This allows for a certain margin of error to be taken into account. More so, when testing for collisions, the *intersect()* method of the Rectangle class returns a new Rectangle representing the collision area. By doing so, it facilitates the correction step, since the algorithm knows exactly how much each figure must be moved.

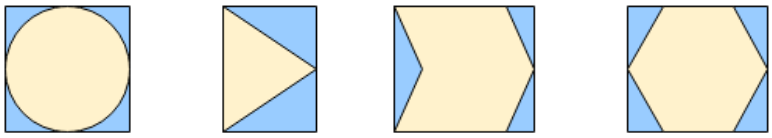

Figure 5.1: Geometric figures and their bounding rectangle

5.1.3 Figure and PolylineConnection classes

Figures are the main units drawn on the canvas by the graphical renderer. A figure can range from the simple outline of a geometrical shape to complex elements which can hold labels (titles), text or even other figures. Custom figures may even be modeled as icons or pictograms, since their background can be coloured, their borders can be stylized and by using extended API even animations could be created.

Polyline connections are wrapper classes which essentially hold a list of points. These points are used to create lines or paths. This is achieved by using the inner list of points to draw Rectangles of negligible height. Just as figures, these lines can be stylized and, in case of representations of directed connections, they can be decorated to emphasize these directions.

The implementation extends both of these classes in order to implement graphical elements with user oriented functionalities. First, the Figure class is extended by the NodeFigure and ModuleFigure classes. A NodeFigure is a figure used when drawing a simple graph for which the user does not provide any special information. It holds and displays only the name of a node. The ModuleFigure, on the other hand, is used to represent a module as it is seen from a HDL designer point of view. It holds information regarding ports, their direction and where they are connected. This type of figure contains and displays labels and text elements which the user can interact with.

PolylineConnection is extended as a OrthogonalConnection. This special type of connection holds references to the figures which it connects and implements a

listener. The role of the listener is to watch for modifications made to the position of the figures. Whenever one changes, the OrthogonalConnections announces the router that it must be recalculated. If an obstacle has been placed on the path in the meantime, the new path shall be computed to go around it.

5.2 Modules implementation

The modules which compose the application, as listed in chapter **??**, utilize a set of Java and OOP concepts and patterns. In this section we shall discuss some of these notable patterns.

5.2.1 Input file format

As was previously presented, the input graphs are read from files written in a certain format, akin to a programming language. This facilitates data manipulation and enables the user to easily describe and understand the model.

The structure of such a file is shown below:

Listing 5.2: Input file example

```
1
2  begin graph
3      name: Example Graph
4          begin node
5              name: A
6              id: 1
7              connections: B
8          end node
9          begin node
10             name: B
11             id: 2
12             connections: A,C
13         node end
14         gebin node
15             name: c
16             id: 3
17             connections: A,D
18         node end
19         begin node
20             name: D
21             id: 4
22             connections: C
23         node end
```

```
24 end graph
```

5.2.2 Parser implementation

The input files parser is implemented using exclusively Java file oriented API. Files are read using a Scanner object, which allows the reading of bytes from the file until certain specified characters are encountered. Upon reaching such a character, the scanner provides a String object containing the read data.

Next, a StringTokenizer splits the String into tokens. Rules and patterns (of the grammar) are applied on these tokens to determine wether they form valid expressions and provide useful information.

5.2.3 Graph data and elements

The graph object itself is constructed using the Factory pattern. Nodes and Edges are both implementations of a GraphElement interface. The graph keeps a list of GraphElement objects which is populate by the parser as the input files are decoded. In order to differentiate between each type of element, the interface exposes the method *getType()*. Each object returns a different value of an enumeration called ElementType.

Using this pattern, the graph object does not have to keep two individual lists, one for each type of element. This helps keep the memory usage lower in case of large amounts of data. When another module requires all elements of a certain type, a new list is constructed on the fly and passed to the caller.

Listing 5.3: Basic class structure for factory implementation

```
1  public interface GraphElement {
2      public ElementType getType();
3  }
4
5  public enum ElementType {
6      VERTEX, EDGE;
7  }
8
9  public class Vertex implements GraphElement {
10     @Override
11     public ElementType getType() { return ElementType.VERTEX;}
12 }
13
14 public class Edge implements GraphElement {
15     @Override
16     public ElementType getType() { return ElementType.EDGE;}
```

```
17 }
18
19 public class Graph {
20     public List<GraphElement> elements;
21     ...
22     public List<GraphElement> getElementsOfType(ElementType type) {
23         List<GraphElement> result = new ArrayList<GraphElement>();
24         for (GraphElement ge : elements) {
25             if (ge.getType.equals(type)
26                 result.add(ge);
27         }
28         return result;
29     }
30 }
31
32 public class Parser {
33     ...
34     public void parse(File inputFile) {
35         ...
36         graph.elements.add(createElement(parsedData));
37         ...
38     }
39     ...
40 }
```

5.2.4 Graph processing modules

The modules which utilize graph data and perform modifications on it: planarity tester, layout processor and edge router have at their core consecrated algorithms. Certain modifications are made to these algorithms in order to properly integrate them in the Java environment. Also, in some cases, elements from different classes of algorithms are combined in the implementation (for example the rule-based placing algorithm).

The planarity tester is a direct implementation of theorems 1 and 2 presented in the Algorithm Design chapter. However, there is also an implementation of Chiba and Nishizieki's algorithm which is used to confirm planarity. This algorithm is called when the necessary condition specified by the theorems holds.

In the case of layout processing, the grid type layout algorithm uses an implementation of the white path theorem algorithm based on DFS(depth first search) to construct the spanning tree used to layout the algorithm. The pseudocode of the white path theorem algorithm is listed below:

Listing 5.4: Spanning tree algorithm using DFS

```
1  PROCEDURE SPANNING-TREE {
2  BEGIN
3      choose arbitrary vertex v;
4      addToSpanningTree(v);
5      WHILE (unvisited vertices)
6      BEGIN
7          IF (v has unvisited neighbours)
8          BEGIN
9              u = chooseNeighbour();
10             markAsVisited(u);
11             addToSpanningTree(edge(v,u));
12         END
13     END
14 END
```

The rule-based layout processor and the edge connection router are direct Java implementations of the algorithms described in the system architecture chapter.

Chapter 6

Results

In the Introduction chapter we presented a set of criteria which the implementation must follow. To be able to evaluate and analyse these metrics, we shall use particular graphs which are shaped to exercise the currently tested module more than the others. In addition to these metrics, we would like to analyze which module performs the most intense computations and how the implementation compares with other similar software.

We must also note that there are criteria such as coherency which are not completely objective. This is a situation where the user decides whether the result is coherent and in accordance with what was expected.

6.1 Planarity testing performance

We have seen that the planarity testing module shall first test if the necessary relationship between edges and vertices is true. This is a constant time - O(1) - operantion. However, should this return a positive result, we can not tell for certain that the graph is planar. The vertex addition method must be applied in order to ensure that the graph is planar. This method has time complexity O(n).

The correctness of the planarity testing can be easily verified by having the algorithm draw the K5(complete graph of 5 nodes) and K4(complete graph of 4 nodes) graphs. K5 is along with K3,3 one of the two basic non-planar graphs from Kuratowski's theorem. K4, however, while may seem not planar, is in reality a planar and must be represented correctly. The resulting drawings are shown in the images below:

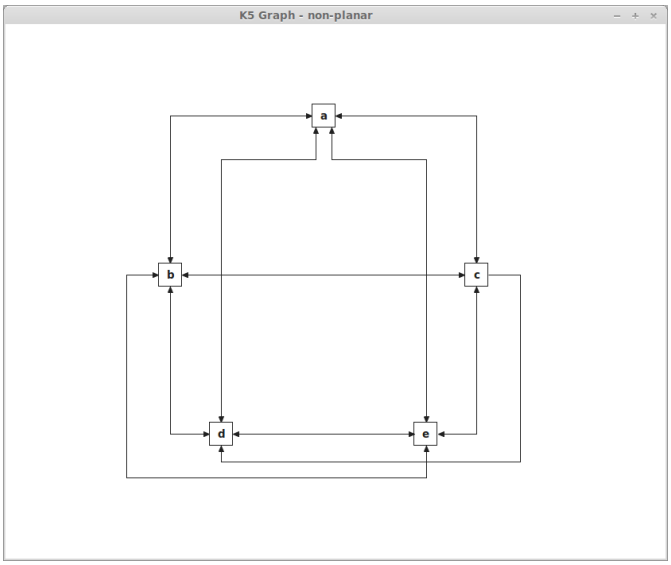

Figure 6.1: The K5 graph. Edges intersect because the graph is not planar.

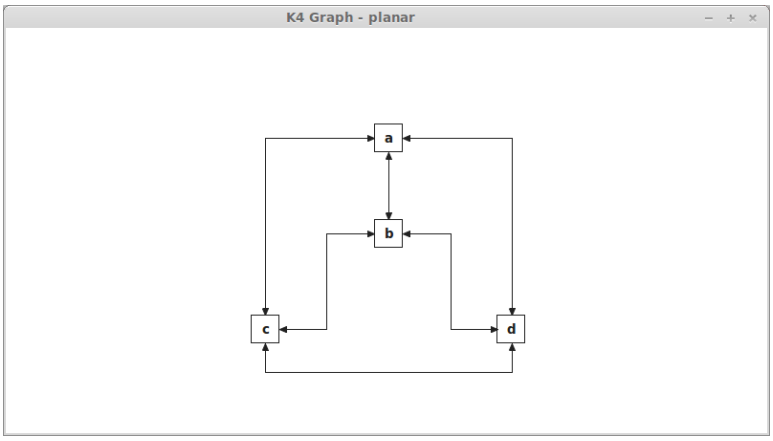

Figure 6.2: The K4 graph. Planarity is preserved.

6.2 Connection routing

Connection routing is, like planarity testing, an operation which has a relatively low time complexity. Firstly, it must travesre all edges at least once and ensure that they are connected. Then, all edges which are correctly routed are removed from the list and the rest are visited once more. The closest approximation for this operation is $O(|E| * \log |E|)$.

Correctness regarding routing means that all connections are orthogonal. This is done by the router module itsefl, because one of the conditions for considering a path correctly routed is it being orthogonal.

6.3 Time consumed per module

Knowing in which module the application spends the majority of the processing time is an important metric to analyze for possible optimizations. Since the modules are mainly independent, we can easily time exactly how much of the processing time is spent in a certain phase.

In order to correctly analyze this metric, we shall consider two different cases. In the first one, the input graph is not planar, while in the second the graph is planar. In both scenarios, the number of nodes in the graph is the same. After running the application multiple times on the same data sets, we obtain the following results:

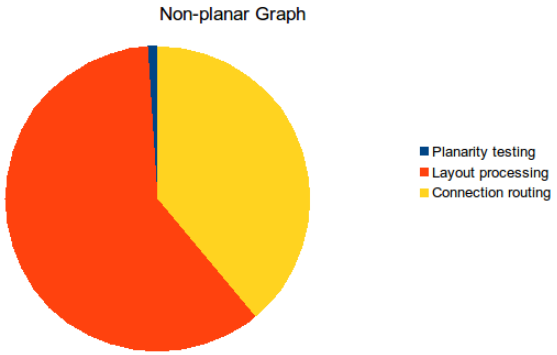

Figure 6.3: Timeing results for non-planar graph test

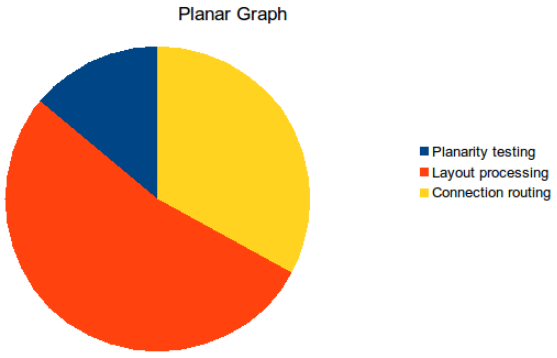

Figure 6.4: Timeing results for planar graph test

We see that in both cases, the most time is spent in the layout processing module. This result is in accordance with our theoretical presumptions: planarity testing is at worst $O(n)$ complexity, and the edge routing is proportional with $O(|E| * \log |E|)$. The complexity of the genetic algorithm surpases both of these.

6.4 Performance testing

To see how well the application performs, we must also compare its performance versus that of existing implementations. The benchmarks used in these comparative tests are the two implementations mentioned in the Related Work chapter: the Graphviz dot command line application and the Eclipse GEF. The benchmarking process is performed on tree structured graphs with increasing number of nodes, rangeing from 10 to 100000 nodes. We analyse only the run time of the computational part of these application, without the GUI enabled.

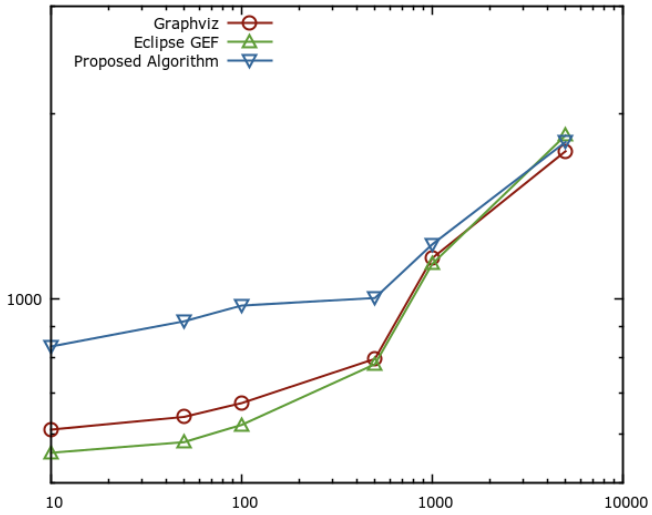

Figure 6.5: Performance benchmark for graphs containing under 10000 nodes. Logarithmic scale is used for both X and Y axis

The above charts show that for smaller graphs, the all the constraint checking and verifications performed by the proposed solution are detrimental to its performance. However, once the number of nodes in the graph starts growing, the approximations used and the policy of prefering the local minimum over the global minimum is beneficial. The point at which the solution becomes favorable is near 10000 nodes.

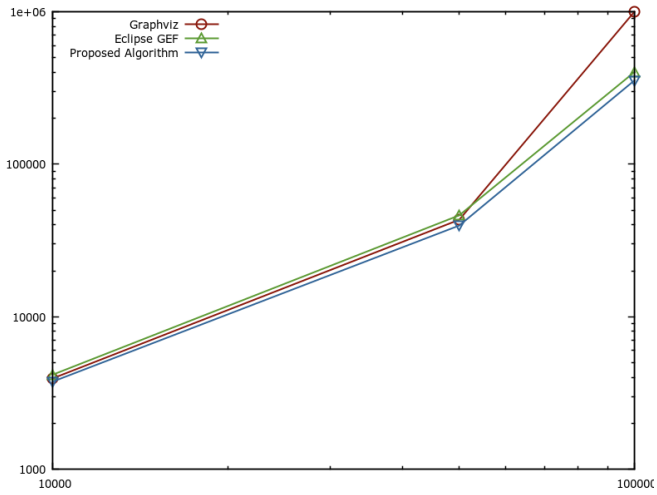

Figure 6.6: Performance benchmark for graphs over containing 10000 nodes. Logarithmic scale is used for both X and Y axis

Chapter 7

Conclusions and Future Work

7.1 Conclusion

This thesis has shown that the graph drawing problem can be approached by combining classical algorithms with modern ones. It is clear that such techniques are more flexible and versatile, making them easier to integrate with various development environments. Another strong point of such an approach is its modularity and portability.

The main contribution of this work was defining and solving a diagram drawing problem that satisfies real user requirements rather than theoretical metrics. An implementation for this solution using an optimized genetic algorithm is also provided. It improves on the areas which are important for users: duration, clarity and correctness. Duration is improved by favouring an approximation instead of the full solution (local minimum vs. global minimum), clarity is assured by choosing the appropriate layout for the presented problem while using orthogonal connectors to draw paths and correctness is determined by testing planarity prior to initiating the routing process.

From the test results, it can be seen that the most important part of such an application consists of the layout processor. Firstly, the majority of the running time is spent in the layout module, which influences performance. Secondly, a performant processor which produces an optimal layout greatly reduces the effort of the routing process and can minimize the canvas area covered by the drawing.

With the increasing interest for integrated development environments in the hardware design and verification world, tools that allow visualisation of various data and concepts are becoming more important. We have shown that by experimenting with different algorithms and integrating concepts from multiple fields of computer science, we can provide an application which fulfills this role. By allowing users to interact and perform modifications, the final drawing can be shaped to look less synthetic and more appropriate for the desired task.

7.2 Future work

Development for the application is set to continue, especially in the direction of usability. The most amount of work needs to be invested in implementing a parallel version of the application. We know that the graph object is used by all three main modules of the implementation, but in a sequential way. It does not truly represent shared data. Because the planarity testing module does not affect the way in which the other two modules run, the first step would be to start planarity testing at the same time as layout processing. We are also aware that the genetic algorithm does not influence the positioning of the central node and its neighbours. In this regard, we could have the edge routing module start operating as soon as this specific set of nodes has been placed. If the layout processor manages to finish placing the other nodes before the routing for these nodes is complete, then routing can continue normally. Otherwise, routing can be halted, though we have still obtained an improvement of the running time.

The design of a dedicated user interface is also important and taken into account for future development. The current look of the application is that of a simple view. This can make the drawings look static from the perspective of the user and does not encourage the idea that they are modifiable. Menus and tool tips can be added for each diagram component to show the user what can be modified. An export function could be useful for users who want to save the drawing in a picture format or save the current configuration and simply re-load it at a later time to avoid having the application run the algorithm on the same set of data, multiple times.

Currently the only unit of hardware design that is supported and represented by the tool is the module. Modules are merely the top level entities and they can contain various other constructs. While displaying relationships and connections between modules is important, users often need to have information about what the modules contain. Otherwise, they represent black boxes whose inner functionality is unknown.

Another point which can be improved is writing and maintaining the input files. The application is integrated with the Eclipse IDE Platform, which allows developers to add functionality by using plug-ins. One such plug-in that can be implemented is a smart editor for input files. A smart editor can offer functionalities such as auto complete, hyperlinks and references, which would make writing and navigating these files a much easier task for users. At the same time, the file parser could be improved to signal syntactic errors on the fly, by placing error markers in such a smart editor.

List of Figures

Listings